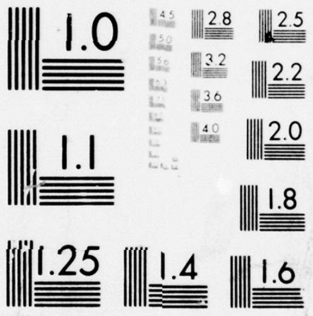
1.0
4.5
5.0
5.6
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

②

RADC-TR-78-157
Final Technical Report
August 1978

AUTOMATED CODE GENERATORS FOR COMPILERS

E. Chu
E. Halb
H. McCoy
R. Morton

Computer Sciences Corporation

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

RADC-TR-78-157 has been reviewed and is approved for publication.

APPROVED: Richard M. Motto

RICHARD M. MOTTO
Project Engineer

APPROVED: [illegible]

WENDALL C. BAUMAN
Chief, Information Sciences Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE 19	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: 18 RADC-TR-78-157 / 2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER: N/A
4. TITLE (and Subtitle): 6 AUTOMATED CODE GENERATORS FOR COMPILERS	5. TYPE OF REPORT & PERIOD COVERED: 9 Final Technical Report, Sep 76 - Mar 78
	6. PERFORMING ORG. REPORT NUMBER: N/A
7. AUTHOR(s): 10 E. Chu, E. Halb, H. McCoy, R. Morton	8. CONTRACT OR GRANT NUMBER(s): 15 F30602-76-C-0429
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Computer Sciences Corporation, 220 Continental Blvd, El Segundo CA 90245	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 63728F; 16 55500843; 17 08
11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (ISIS), Griffiss AFB NY 13441	12. REPORT DATE: 11 August 1978
	13. NUMBER OF PAGES: 18; 12 18p.
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Same	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION DOWNGRADING SCHEDULE: N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer:
Richard M. Motto (ISIS)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

JOVIAL
Compilers
Code Generators

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This project was undertaken with the intent of devising a method or tool for automating the production of the code generator portion of compilers. The main objective was to develop a Quick Code Generator (QCG) that would be capable of producing JOVIAL compilers in less than 3 months. This generator would develop "loose" code for an interim compiler. This effort resulted in developing the Machine Independent Macro Instruction Code (MIMIC) language. This language consisted of 113 macros & a QCG was successfully produced for the (Cont'd)

next page

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

S/C 094 130

20 (Cont'd)

HIS 6000 system. However, the effort fell short by not producing macro definition files for other target machines. It should be noted that RADC's JOVIAL Compiler Validation System was executed with the original JOCIT compiler & the QCG JOCIT compiler & the differences in compilation time, execution time & the total code produced were insignificant.

and

Evaluation

1. This effort, entitled "Automated Code Generators for Compilers" is an attempt to produce a Quick Code Generator (QCG) for an interim compiler which will permit the use of this compiler until a permanent, more efficient code generator is developed. The output of the QCG is assembly language for the target machine rather than the object code normally associated with code generators. RADC's JOCIT JOVIAL Compiler was the subject of this effort and the approach taken was to replace the JOCIT Code Generator and Editor with a QCG and Macro Assembler (MAS), respectively. Communication between these two phases was accomplished via the macro language, MIMIC. MIMIC is an acronym for Machine Independent Macro Instruction Code and is a generalized assembly-type language, designed to easily map into most existing assembly languages. The user is required to write a Macro Definition File (MDF) that is to interface with MIMIC.

2. This effort was successful in developing a QCG for JOCIT for the HIS 6000 Series, but several problems are still to be resolved. Mainly, the QCG will not operate with the JOCIT Optimizer, nor will the compiler function correctly if a cross-reference listing is requested or the DIRECT Code Option is used. Problems still exist with internal data representations, but it should be noted that tests comparing the original JOCIT

JOVIAL Compiler and the QCG JOCIT Compiler revealed the difference in compilation time, execution time and total code produced was insignificant.

Richard M. Motto

RICHARD M. MOTTO
Project Engineer

INTRODUCTION AND SUMMARY

The "Automated Code Generators for Compilers" contract F30602-76-C-0429 was begun in September, 1976. It was one of a series of contracts to provide the Air Force with an advanced JOVIAL (J3) compiler capability. Earlier contracts have resulted in a system called JOCIT which is an acronym for JOVIAL Compiler Implementation Tool. JOCIT consists of a JOVIAL compiler, a SYMPL compiler, GENESIS processor and an executive which interfaces with the host operating system (in this case, Honeywell 6000 GCOS). Most of the modules of the JOVIAL compiler and the SYMPL compiler are written in the SYMPL language. SYMPL is a procedure-oriented programming language whose emphasis is on ease of expression of "System" programs. SYMPL is in many ways similar to both FORTRAN and JOVIAL. GENESIS is a Computer Sciences Corporation proprietary system for producing language syntax tables suitable for use in certain types of syntax-directed compilers from a convenient syntax description language.

The primary purpose of this contract was to further automate the task of producing a JOCIT-built compiler code generator. There was no expectation that the code generator produced by this effort would equal the quality or efficiency of the code generators produced by an experienced compiler writer. However, if the result of this effort could rigidly and inexpensively produce a code generator of acceptable (not defined) quality, then a significant advance would be made to the state-of-the-art. The code generators thus produced would presumably be used temporarily in new operational environments until a more permanent, efficient code generator was written for the particular target computer.

A number of approaches were investigated, but the final design consisted of the "code generator" producing macros for an assumed very basic computer. For example, the computer was assumed to have one accumulator, one index register, one single precision floating point register, etc. The macro code file thus generated is machine independent or nearly so. This macro code file is then processed by the "macro assembler" which essentially takes the place of the editor in a conventional compiler. The "macro assembler" itself is also machine independent but accesses a machine-dependent "macro definition file". Each new target machine requires the production of its macro-definition file.

The macro definition file essentially defines each of the macros which can be produced by the "code generator" in terms of assembly language attributes of the target machine. The macro assembler, then, as it processes the macro code file, produces a new output file of a target machine assembly language program. This assembly language program can then be assembled on the target machine to produce object code for execution.

Therefore, in order to produce a new "code generator" for a target machine, an individual would have to have a good working knowledge of the assembly language of the target machine and the purpose for each of the macros that are possible. Currently, the "code generator" can produce approximately 113 macros. However, many are similar in nature and would require only minimal changes in their definitions.

The final result of this contract is a working "quick code generator". A macro definition file for the Honeywell 6000 series computer was written and several test programs were run through the new code generator. Assembly language programs were produced and subsequently assembled by the GMAP assembler at RADC and executed. There was not sufficient time to generate macro definition files for other possible target machines. However, it is believed that a person familar with a target machine and some study of the Automated Code Generator User Manual could produce a macro definition file in less than three months.

The "quick code generator" was integrated into JOCIT, so that one JOVIAL compiler can use either the "quick" or the "normal" code generator. This is accomplished by the use or non-use of the QUICK option in the JOVIAL control card. It is also possible to select four different macro definition files, if they are available, to produce code for different target machines.

In order to complete a working version of the new code generator in the time available, a number of features permitted with the normal code generator are not possible currently with the "quick" code generator. These include "direct code" and global optimizations. Furthermore, the cross-reference capability cannot be correctly executed. During the course of this contract a parallel effort was performed on JOCIT to add double precision capability. This complicated the effort on the "quick" code generator, but the double precision capability is operational now for the "quick" option as well.

AUTOMATED CODE GENERATORS FOR COMPILERS

The purpose of the JOCIT Automated Code Generator (ACG) is to help automate the retargeting of the JOCIT compiler. The ACG provides a means for the rapid development of a new compiler "back end", producing code for a desired target machine. The goal is to make a compilation facility available early, so that a user considering the use of the JOVIAL High Order Language (HOL) will not have to wait up to a year for the handcrafting of a code generator to fit his needs. If a high degree of code efficiency is important, then a special code generator has to be constructed, but at least a coding and checkout environment will be available through the ACG several months before the final code generator is ready.

The first design decision made for the ACG was to output assembly language code for the target machine's assembler, rather than to produce binary code. The reason for this decision was the elimination of the target machine's relocatable format and linker requirements from consideration in the retargeting effort, as well as internal instruction and data representations.

STRUCTURE OF THE ACG

Structurally, the ACG consists of (1) a modified version of the Code Generator phase, called the Quick Code Generator (QCG), (2) a new phase in lieu of the Editor phase, called the Macro Assembler (MAS), and (3) a group of procedures supplied by the retargettor, called the Macro Definition File (MDF).

QUICK CODE GENERATOR

Providing a suitable set of MDF procedures is the principal effort of retargetting; each of these procedures has a specified task in the production of the assembly code output. In determining what these tasks should be, many computers and their assembly languages were studied. Among the architectural features considered were addressibility, register usage, word sizes, and alignment requirements. Assembler considerations included formats, naming conventions, special features, pseudo-ops, etc. The result of these studies was the MIMIC (Machine Independent Macro Instruction Code) "language", which drives the generation of the output assembly code. MIMIC is the internal language (IL) form of the program produced as output by the QCG. Essentially, it is "machine code" for a hypothetical and very simple computer. The architecture of the MIMIC computer has been carefully chosen to avoid presenting any impossible situations in representing its functions on a real computer. The MIMIC machine has the following characteristics:

- Single address instruction format
- One fixed point or integer accumulator register
- One single precision floating point accumulator register
- One double precision floating point accumulator register
- One index register
- Access to all types of data
- Arithmetic operations in integer, single floating, and double floating modes
- Comparisions in integer, single floating, and double floating modes
- Full word addressing with optional index register address modification
- Although different MIMIC instructions are used to load each of the three accumulator registers, the QCG assumes that these registers overlap, so that a load of any one of them destroys the contents of the others.

The full MIMIC language is documented elsewhere, but an example, is appropriate here. A simple JOVIAL replacement statement might produce the following sequence of MIMIC instructions:

```
LOAD     operand AA        (Load the integer accumulator)
SCLL       2               (Scale left by 2)
ADDI     operand BB        (Add to the integer accumulator)
STORE    operand CC        (Store the integer accumulator)
```

The QCG generates into the Code File a sequence of MIMIC instructions representing the executable part of the source program, together with MIMIC pseudo-ops declaring data reservations, external references and definitions, etc., obtained by scanning the JOCIT Symbol Table.

The task of the retargettor is to determine, for each MIMIC "macro" (instruction or pseudo-op), what lines of assembly code should be produced for the target machine. This includes the related design tasks of determining how to use the target machine's registers, what linkage conventions will be followed in procedure calls, how the target machine assembler's location counters will be used, etc. Then the user must supply, in the MDF module, a procedure entry point for each MIMIC macro.

An early design decision was made to have these MDF procedures coded in SYMPL. The alternative was to provide a new language, with a preprocessor to convert the new language into an interpretable form. A new language was indeed designed and studied, but was rejected on two grounds. First, its advantages over SYMPL as a coding language were not felt to be striking enough to justify the effort and expense of producing an additional processor and interpreter. Second, it was a less powerful language than SYMPL, and it was not clear that anything less powerful than a full algorithmic language could be guaranteed to solve any unforseen problems which might arise in a specific retargetting effort. Using SYMPL, the MDF programmer is free to have global variables which pass information from one MIMIC macro procedure to another, as well as lower-level procedures for common functions.

In deciding how to implement the Quick Code Generator (QCG), several alternative approaches were considered for various aspects of the design. The first, and most critical alternative concerned the overall approach. Here, the options were:

1. Write a completely new code generator, or
2. Create new modules using existing code generator modules.

CSC's decision was to use the existing modules as a base on which the QCG would be built, since this approach (1) would yield a code generator which was very similar to the existing code generator, (2) would save considerable time, and (3) would simplify maintenance. In making this transformation, several previously-undetected errors in the compiler were found and corrected.

A second major consideration concerned the handling of direct code. The options considered were:

1. Modifying all of the current modules which handle direct code processing.

2. Prohibit the inclusion of direct code by the QCG.

In arriving at our decision to prohibit direct code, we took into consideration several factors:

1. In any retargetting effort, the direct code must be rewritten for the target machine and could, therefore be called as an external subroutine instead of appearing in line.

2. The existing programs which process the IL would require modification to pass source code through to the QCG.

3. The new optimizer under concurrent development would have been impacted.

4. The total benefit as weighed against the implementation time required was minimal.

The QCG contains a set of predetermined code sequences for producing frequently used functions. For example, the JOVIAL statement A = B could cause a LOAD macro and a STORE macro to be generated. The code sequences are table driven and register dependent. For example, given the statement A = B + C, if either B or C already exists in a register based on a previous computation, the QCG recognizes the fact and, instead of loading the register, uses the already computed value to generate the ADD and STORE instructions. The type of register to be used is one of the parameters passed; therefore, only one sequence exists, and it is valid for integer, single precision, and double precision data. This single code sequence of the QCG is a departure from the regular code generator which uses a separate set of sequences for integer, single precision, and double precision data. The processing of character data is also slightly different from the regular code generator. The MIMIC language does not provide for string manipulation macros and, therefore, all character data is handled through subroutine calls to the run-time library. This approach required that a set of arguments be generated to adequately define the data for the subroutine so that the subroutine could operate properly. This approach was taken to relieve the macro writer of the burden of developing string handling macros. Such macros can be come quite complex when implemented on word oriented hardware. To facilitate the retargetability of these subroutines, they were written in JOVIAL. The regular code generator produced in-line code, when possible, to handle character data. When this was not possible, subroutine calls were generated.

Near the end of the project, it was discovered that some test cases would not compile correctly through the ACG if the Optimizer was requested. Rather than spend the short time remaining on solving the problem, it was decided that optimization would not be allowed if the ACG was to be run. The two major reasons for this were: first, the new optimizer being developed had not yet been interfaced with the ACG and similar problems would likely appear there as well, causing further delays in the project; second, given the limited scope of use of the ACG (i. e., as a "stop gap" measure during the writing of a target machine dependant code generator), and given the limited characteristics of the hypothetical machine upon which MIMIC is based, optimization would have little or no major affect on the code produced. Additionally, the quality of code is to a great extent, dependent upon the level of expertise of the macro writer.

Because of its more general design the QCG generates more temporary space for the computation of intermediate results. The management of this space is critical in keeping the size of this temporary space to a minimum. Also, failure to release temporary space when no longer needed often results in compiler errors, incorrectly allocated space, or both. Persons maintaining this compiler must be aware of this critical area so that compiler errors are not introduced.

MACRO ASSEMBLER PHASE

The MAS phase consists of two levels: a driver for the MDF procedures, and a group of utilities to serve the MDF procedures. The driver level, after initialization, reads the MIMIC macros from the Code File in sequence and, for each, invokes the corresponding MDF procedure, passing the MIMIC operand as an argument. In the initialization step, the MDF programmer has an opportunity to specify a number of parameters of his assembler, e.g., the character which introduces a comment, the maximum name length, etc. This information will be used by the MAS utility procedures. The MDF programmer can also specify the order in which he wants to process the various segments of the program, such as code, local data, external declarations, etc.

The MAS utilities available to the MDF programmer fall into three groups. The first group consists of editing procedures which assist him in building his output lines by performing such functions as adding a string of characters to a partially built line, tabbing to a specified position in a line, and outputting a line and reinitializing. These utilities handle most of the work of character manipulation and line formatting.

The second group of utilities gives the MDF programmer access to information in the JOCIT Symbol Table, such as the allocated size of an item, or whether or not an item is signed. The third group provides such miscellaneous services as generating unique labels for use if local branches are needed.in the assembly code.

The MAS also provides error checking to assist in the debugging stage of a new MDF. The editing utilities detect such errors as runaway line length, and print the current output line. The symbol table utilities trap an invalid symbol table pointer and identify the called procedure. An error limit may be supplied by the user, to stop processing after a specified number of MAS errors have been detected.

As a test of the system, an MDF module was designed and implemented to produce code for the current host machine. A number of programs were compiled, assembled by GMAP, and successfully executed. However, as noted later, some problems remain even with this "retargetting".

A programmer assigned to the MDF portion of a retargetting effort must be very familiar with his target machine, its assembler, and its various system conventions. He must also have a working knowledge of the SYMPL language, in which he will be coding. Then, he must understand the meaning and intent of each MIMIC macro and its operand, which may involve some understanding of JOVIAL semantics (although an understanding of JOVIAL syntax is not required). Finally, he must become familiar with the MAS interface and the various utilities at his disposal.

The other major effort in retargetting JOCIT is converting the JOVIAL Run-Time Library. Investigation has shown that the non-I/O library modules can successfully be written in JOVIAL, and thus are retargettable like any other program. Simple machine dependencies, such as word and byte size, are handled by JOVIAL DEFINE declarations, and must be changed in the source to suit the target machine characteristics. However, the I/O package is so strongly permeated with system and machine dependencies that it requires a complete recoding in assembly language for each new target.

A number of problems encountered during the performance of the work remain unsolved. This situation is due in large part to the schedule constraints, and the fact that several other JOCIT enhancement activities were going on concurrently, some with higher priorities than the ACG. These problems were mainly in the area of incompleteness of JOCIT conversion.

The QCG part of the ACG is essentially an effort to retarget the JOCIT code generator to the hypothetical MIMIC machine. To a large extent this was successful. However, the interactions between the code generator and certain other non-essential (but certainly useful) compiler functions have not been resolved. Specifically, the QCG cannot be run with the Optimizer, nor will the ACG compiler function correctly if a map or cross-reference listing is requested. Although in principle it should be even easier to handle target machine DIRECT code when the compiler itself outputs assembly language, the necessary modifications have not been made, and the ACG compiler will not accept DIRECT code.

Unfortunately, a successful retargetting of JOCIT requires more than Code Generator work. A specific unsolved problem area outside the scope of the Code Generator is the question of internal data representations. The current "front-end" of the compiler converts numeric constants, such as preset values, to Honeywell machine forms. On the one hand, this seems only reasonable if the compiler, for example, is to do constant arithmetic on the host machine. On the other hand, however, considerable difficulties arise in the MAS and MDF in trying to go from a Honeywell bit pattern to a meaningful source-language constant in a foreign assembly language.

Differences in word size may also invalidate some of the compiler's host-oriented conversions and operations. This area clearly needs much investigation to produce a viable solution.

To facilitate retargetting and rehosting efforts in the future, the compiler should be carefully examined and generalized parameters should replace all machine dependent values currently hard coded in the programs.

IED

-78